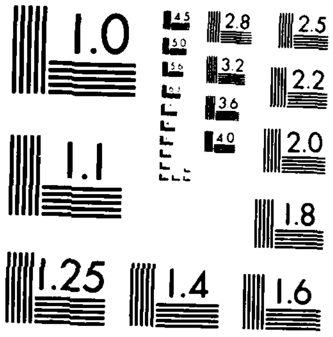
1.0
4.5
5.0
5.6
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

(4)

CAR-TR-57
CS-TR-1388

F-49620-83-C-0082
May, 1984

Dynamic Programming and Transitive Closure on Linear Pipelines

I.V. Ramakrishnan
Department of Computer Science
University of Maryland
College Park, MD 20742

P.J. Varman
Department of Electrical Engineering
Rice University
Houston, TX 77001

COMPUTER VISION LABORATORY

CENTER FOR AUTOMATION RESEARCH

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

CAR-TR-57
CS-TR-1388

F-49620-83-C-0082
May, 1984

Dynamic Programming and Transitive Closure on Linear Pipelines

I.V. Ramakrishnan
Department of Computer Science
University of Maryland
College Park, MD 20742

P.J. Varman
Department of Electrical Engineering
Rice University
Houston, TX 77001

Abstract

Algorithms for the dynamic programming and transitive closure problems are presented for a linear pipeline of processors. These algorithms require only a *constant* number of ~~I/O~~ input/output ports and are *optimal* in their area and time requirements.

A-1

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 84-0559
6a. NAME OF PERFORMING ORGANIZATION: University of Maryland; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: Air Force Office of Scientific Research
6c. ADDRESS (City, State and ZIP Code): Center for Automation Research, College Park MD 20742	7b. ADDRESS (City, State and ZIP Code): Directorate of Mathematical & Information Sciences, Bolling AFB DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR; 8b. OFFICE SYMBOL (If applicable): NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F49620-83-C-0082
8c. ADDRESS (City, State and ZIP Code): Bolling AFB DC 20332	10. SOURCE OF FUNDING NOS.: PROGRAM ELEMENT NO. 61102F; PROJECT NO. 2304; TASK NO. A2; WORK UNIT NO.

11. TITLE (Include Security Classification): DYNAMIC PROGRAMMING AND TRANSITIVE CLOSURE ON LINEAR PIPELINES.

12. PERSONAL AUTHOR(S): I.V. Ramakrishnan and P.J. Varman

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Technical	FROM ______ TO ______	MAY 84	32

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Computer architectures; VLSI; algorithms; dynamic programming; transitive closure; pipeline processors.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Algorithms for the dynamic programming and transitive closure problems are presented for a linear pipeline of processors. These algorithms require only a constant number of I/O ports and are optimal in their area and time requirements.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED [✓] SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Robert N. Buchal	22b. TELEPHONE NUMBER (Include Area Code): (202) 767- 4939; 22c. OFFICE SYMBOL: NM

DD FORM 1473, 83 APR   EDITION OF 1 JAN 73 IS OBSOLETE.   UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

84 07 24 050

1. Introduction

Dynamic programming and transitive closure are two important computational problems. Dynamic programming is one of several widely used problem-solving techniques in computer science and operations research (see Brown's review in [4]). The transitive closure algorithm also arises in many contexts. For example, in the data-flow analysis of programs, we often need the closure of the "call" relation.

Straightforward dynamic programming requires $O(n^3)$ [1] sequential time where n is the problem size. Similarly, well-known serial algorithms for transitive closure of an $n \times n$ matrix require $O(n^3)$ time [19,20]. As matrix multiplication and transitive closure are computationally equivalent [1], the time complexity of the transitive closure algorithm can be further reduced by the methods of Pan [12]. However, the best known upper-bound on the time complexity for matrix multiplication is $O(n^{2.79})$ [12] which is achieved at the expense of complicated code.

Parallel algorithms for these two problems have been studied in the past [6,11,17]. The best known upper bounds on the parallel time complexity for these two problems is $O(n)$ reported by Guibas et al. [6]. They use a systolic array of $O(n^2)$ processors.

Systolic arrays (see [9] for a description of systolic arrays) have been proposed as a simple and effective means of employing VLSI technology to handle compute-bound problems. These array processors are typically made up of simple, identical processing elements (which we will refer to as *cells* from now on) that operate in synchrony. Several array structures have been proposed that include linear arrays, rectangular arrays and hexagonal arrays. High performance is achieved by extensive use of pipelining and multiprocessing. In a typical application, such arrays would be attached as peripheral

[1] $f(n) = O(g(n))$ and $f(n) = \Omega(h(n))$ if there exists constants c_1, and c_2 such that $f(n) \leq c_1 g(n)$ and $f(n) \geq c_2 h(n)$ respectively.

devices to a host computer which inserts input values into them and extracts output values from them.

In practice linear arrays are more attractive than two-dimensional arrays (like a mesh and a hexagonal array). Among them are the following: Linear arrays have bounded I/O requirements [9]. In a wafer containing faulty cells, a large percentage of non-faulty cells can be efficiently reconfigured into a linear array [10]. Synchronization between cells in a linear array can be achieved by a simple global clock whose rate is independent of the size of the array [5].

In this paper we present linear array algorithms for dynamic programming and transitive closure problems. Our algorithm uses $O(n)$ cells and requires $O(n^2)$ time steps for dynamic programming problems of size n and transitive closure of $n \times n$ matrices. $O(n^2)$ time steps is *optimal* as at least n^2 time steps are needed to insert the elements in the array. Each of the cell in the array requires $O(n)$ storage (referred to as *area* in the VLSI context). We will show that $O(n^2)$ storage used in the array is *optimal.*

Parallel algorithms for these two problems that have appeared in the past are vulnerable to failures in the cells and communication links in the parallel architectures on which they run. This is very likely in the systolic array solution proposed by Guibas et al. Systolic arrays implemented in VLSI can have (with high probability) faulty cells and links caused by production faults in the manufacturing process that result in defects occuring randomly in the wafer [2].

Varman and Fussell [18] presented a technique to transform "one-way" pipelined linear-array algorithms (that is, algorithms wherein elements in the linear array move only from left to right) into an equivalent algorithm on any connected component of cells by configuring it into a logical-linear array. Neighbouring processors need not be physically adjacent in the connected component. The connected component of cells could form the non-faulty cells in an underlying network that has both faulty and non-faulty cells and communication links. As we will see later on, our algorithms for dynamic programming and transitive closure are one-way pipelined algorithms and hence can be made robust by straightforward application of the technique in [18].

The remainder of this paper is organized as follows. In Sections 2 and 3 we describe our algorithms for dynamic programming and transitive closure respectively. In the appendix we provide proofs of correctness of these algorithms and also establish the optimality of the area required by the array.

2. Dynamic Programming

Many problems can be solved by the use of dynamic programming techniques. In order to describe our array algorithm without excessive generality, we will focus on the construction of an optimal binary search tree which is a well-known example of dynamic programming. An optimal binary serach tree is constructed by computing the following recurrence (see Knuth [8] for details):

$$c(i,j)=w(i,j) + \min_{i<k<j} \{c(i,k)+c(k,j)\},\ 1\leq i<j\leq n+1$$

We compute this recurrence on a linear array of n cells. The array is comprised of four data belts - H_f, H_s, V_f, and V_s; two control belts (each 1-bit wide)-H_c and V_c and an address belt A_d as shown in Fig. 2.1. below.

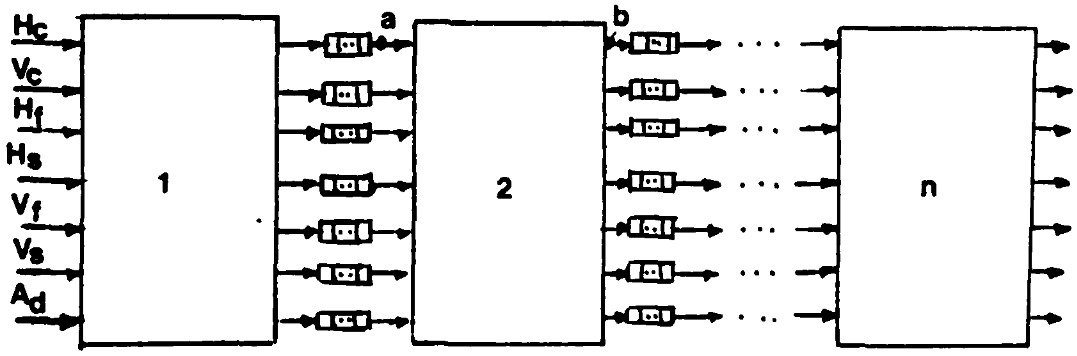

Figure 2·1

Tokens are inserted into these belts at the input of cell 1 and emerge from the output of cell n. The tokens stay on the same belts as they traverse the array. The tokens travelling in H_c, V_c, H_f, H_s, V_f, V_s, and A_d encounter a delay of 4, 2n+3, 2, 4, 2(n+1), 2(n+2) and 2 clock cycles respectively between any cell i and i+1. These delays can be implemented by shift-registers. " " in the figure above denote shift-registers on a belt between cells. A token enters a cell from the left (its input) in the beginning of a cycle and emerges from the right (its output) at the end of the cycle (possibly updated). For example, tokens on H_c enter cell 2 at *a* and leave at *b*. We will refer to the tokens at a cell's input as its *input* tokens.

Each cell in the array has a local memory of size n. The operation of a cell in any clock cycle then is the following. Let x be the contents of the address token at the cell's input. The cell updates location x in its local memory. The new value of x in its local memory is the minimum of the old value, the sum of the contents of its input tokens on H_f and V_s and the sum of the contents of its input tokens on H_s and V_f. If the control bit is set in its input control token on belt H_c then it changes the contents of its input tokens on belts H_f and V_f to the updated value of location x. Lastly, if the control bit is set in its input control token on belt V_c then it changes the contents of its input tokens on belts H_s and V_s to that of its input tokens on H_f and V_f respectively. The linear-array algorithm then is the following.

1. Store w(i,j) in cell j-i at location n-i.

2. At cell 1 do the following:

   a. Insert a control token on H_c with its control bit set at time 2kn+2, $\forall k \geq 0$.

   b. Insert a control token on V_c with its control bit set at time 2kn+1, $\forall k \geq -n$.

c. Insert an address token initialized to adrress k on belt A_d at time $2(kn+1+l)$, $\forall$ $k \geq 0$ and $\forall l \mid 0 \leq l \leq n$.

This completes the description of the algorithm. The effect of the algorithm is the following. Let δ=n-i and γ=j-i. Let c(i,j) denote the token in location δ of cell γ that is initialized to w(i,j) and eventually transferred onto H_f and V_f.

c(i,j) is computed and ready in cell γ at time $2[\delta n+1+2(\gamma-1)]$. The cell then starts transmitting *c*(i,j) on both H_f and V_f. c(i,j) travels on H_f for an additional 2γ clock cycles and is then transferred onto H_s at cell 2γ. It then remains on H_s till eternity. Analogously, c(i,j) travels on V_f for an additional $2\gamma(n+1)$ clock cycles before being transferred onto V_s at cell 2γ whereupon it travels on V_s till eternity.

Example: Consider computation of c(1,5) where n=4.

Now c(1,5)=w(1,5)+min {c(1,2)+c(2,5), c(1,3)+c(3,5), c(1,4)+c(4,5) }.

c(1,3) and c(3,5) are ready in cell 2 at time 30 and 14 respectively. c(1,3) then travels on H_f for an additional 4 (γ=2) cycles and reaches cell 4 (2γ=4) at time 34. c(3,5) travels on V_f for an additional 20 ($2\gamma n+2\gamma$=20) cycles and reaches cell 4 at the same time. From step (2b) of the algorithm the control token inserted at time 1 reaches cell 4 at time 34 (the delay on V_c is 2n+3 cycles/cell). So at time 34 then c(3,5) is on both V_f and V_s and c(1,3) is on H_f and H_s (recall the cell operation when a control token on V_c is present at its input).

c(2,5) is ready at time 26 in cell 3. It travels on V_f from cell 3 and arrives at cell 4 at time 36 (the delay on V_f is 2n+2=10 cycles/cell). The case of c(1,2) is interesting. It is ready in cell 1 at time 26. It then travels an additional 2 clock cycles on H_f till it reaches cell 2 ($\gamma=1$) at time 28. It is then transferred onto H_s. It travels on H_s for an additional 8 cycles (the delay on H_s is 4 cycles/cell) till it reaches cell 4 at time 36. At time 36 c(1,2) and c(2,5) arrive on H_s and V_f respectively at cell 4. Similarly it can be verified that c(1,4) and c(4,5) also arrive at 4 on H_f and V_s respectively at time 36.

3. Transitive Closure Algorithm

The transitive closure algorithm is the following (see [1] for details). Consider an $n \times n$ matrix A of 0's and 1's. This boolean matrix can represent a directed graph, if we let the vertices of the graph be 1,2,..,n and the element a_{ij} of the matrix be 1 if there is an edge from i to j and 0 otherwise. The *transitive closure* A^* of A is also a boolean matrix where the $(ij)^{th}$ entry (denoted as a_{ij}^*) is a 1 if and only if there is a directed path from vertex i to vertex j in the graph. By definition every vertex has a path to itself.

Let a_{ij}^* denote a k-path from vertex i to vertex j that passes through no vertex numbered higher than k except the end points. The transitive closure then can be evaluated using the following recurrence (see [1] for details):

$a_{ij}^{(k+1)} = a_{ij}^{(k)} \vee (a_{ik}^{(k)} \; a_{kj}^{(k)}),\ 1 \leq i,j,k \leq n$

We compute this recurrence on a linear array of 2n-1 cells. The array is comprised of two data belts - H_f and V_f; two control belts (each 1-bit wide) - H_c and V_c and an address belt A_d as shown in Fig. 3.1. below.

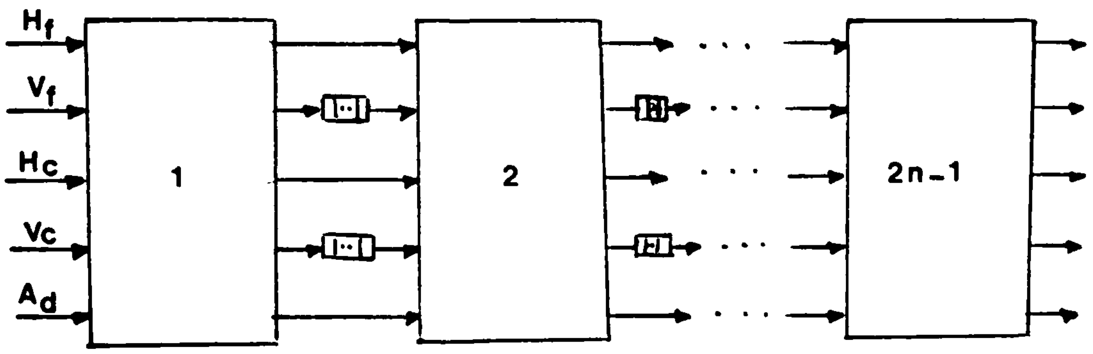

Figure 3·1

Tokens are inserted into these belts at the input of cell 1 and emerge from the output of cell 2n-1. As in the algorithm for dynamic programming, these tokens stay on the same belts as they traverse the array. The tokens travelling on H_c, V_c, H_f, V_f and A_d encounter a delay of 1, (n+1), 1, (n+1) and 1 clock cycles between any cell i and i+1. A token enters a cell from the left (its input) at the beginning of a cycle and emerges from the right (its output) at the end of the cycle (possibly updated).

Each cell in the array has a local memory of size n. The operation of a cell in any clock cycle then is the following. Let x be the contents of the address token at the cell's input. The new value of x is the old value that is ORed to the ANDed contents of its input tokens on H_f and V_f. If the control bit is set in its input control token on belt H_c then it changes the contents of its input token on belt V_f to the updated value of x and if the control bit is set in its input control token on belt V_c then it changes the contents of its input token on belt H_f.

Our linear array algorithm is a three-pass one. We use two copies of the matrix A. Let a_{ij} denote the $(ij)^{th}$ entry in one copy and a_{ij}' denote the same entry in the other copy. Although initially a_{ij} and a_{ij}' are the same in both the copies, these values change as the algorithm progresses. a_{ij} travels on H_f and a_{ij}' travels on V_f.

Let c(i,j) denote the token in location i of cell i+j-1. Let t_s^p $(1 \leq p \leq 3)$ denote the time when a pass begins. The linear array algorithm is the following.

1. Begin the first pass at t_s^1, the second pass at $t_s^2 = t_s^1 + (2n-1)(n+1)$ and the third pass at time $t_s^3 = t_s^1 + 2(2n-1)(n+1)$.

2. In every pass p $(1 \leq p \leq 3)$ do the following at cell 1.

   a. Insert a_{ij} on H_f at time $t_s^p + n(n-1) + n(i-1) + (j-1)$.

b. Insert a'_{ij} on V_f at time $t_s^p+(n-j)n+(i-1)$.

c. Insert a control token with its bit set on V_c when a'_{ii} is inserted on V_f.

d. Insert a control token with its bit set on H_c when a_{ii} is inserted on H_f.

e. Insert address i on A_d when a_{ij} is inserted on H_f.

This completes the description of the algorithm. At the end of the three passes c(i,j) will have a 1 if and only if the transitive closure of matrix A has a 1 in that position.

Example: Consider the graph shown in Fig. 3.2 below comprised of four vertices.

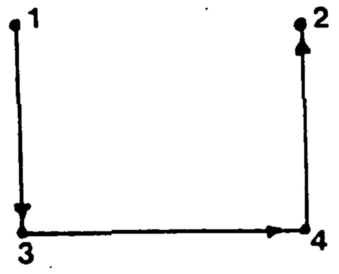

Figure 3·2

We illustrate the computation of a^*_{12}. In pass 1, a_{13} and a'_{34} (which are both initialized to 1) are inserted at times $t_s^1 + 14$ and $t_s^1 + 2$ respectively. a_{13} and a'_{34} meet at cell 4 at time $t_s^1 + 17$ (a'_{34} travels on V_f which has a delay of 5 cycles/cell). So c(1,4) in cell 4 is set to 1 at time $t_s^1 + 17$. a_{14} is inserted at time $t_s^1 + 15$. It reaches cell 4 at time $t_s^1 + 18$ whereupon it is set to 1.

In the second pass, a_{14} and a'_{42} (which are initialized to 1) are inserted at times $t_s^2 + 15$ and $t_s^2 + 11$. They meet at cell 2 at time $t_s^2 + 16$ whereupon c(1,2) is set to 1.

4. Concluding Remarks: We have presented a linear array algorithm for dynamic programming and transitive closure problems that are *optimal* in their *area* and *time* requirements. Our algorithms are suitable for realization in VLSI. Using the technique in [18] our algorithms can be made to run on several parallel architectures, like tree machines [15] and mesh arrays, that have faulty cells.

Realizing the algorithms in VLSI raises some practical issues. In particular, we will consider the potential mismatch between the size of the problem being solved (k) and the size for which the chip is handled (n). If $k>n$, the problem can be partitioned into blocks of size n and the chip used iteratively to handle each block. An obvious solution to handling the case when $k<n$ is to consider the problem of size k as part of a bigger problem of size n (obtained by padding the problem of size k with dummy elements). This would however result in a time penalty factor of $(\frac{n}{k})^2$ over that obtained by using a chip of compatible size. An alternative approach is to configure the chip, as a preprocessing step, to match the problem size. This would require decreasing the number of cells configured to k and decreasing the size of the buffer in each cell appropriately. The selection of cells can be efficiently accomplished on a reconfigurable network such as the CHiP [14]. Changing the buffer size requires the shift registers implementing the buffers to have variable lengths, similar to the proposal in [3]. However requiring the shift register length to be continuously variable (that is, for all values of k from 1 to n) would be prohibitively expensive in terms of layout and area complexity. The algorithms can be modified to run on k cells without changing the buffer size (details omitted in this paper). These modified algorithms have a time complexity of O(nk) and hence this results in a time penalty factor of $O(\frac{n}{k})$. Let the buffer of size N be divided into α equal partitions, which can be tapped at $\frac{Nm}{\alpha}$, $\alpha \geq m \geq 1$. Then a problem of size k,

$\frac{N(m-1)}{\alpha} < k \leq \frac{Nm}{\alpha}$, will employ a buffer of size $n = \frac{Nm}{\alpha}$. The time penalty factor in such a case will be $O(\frac{Nm}{\alpha k})$. It is seen that $\forall k \geq \frac{N}{2\alpha}$, this factor will never exceed 2. This implies, for example, that with just four partitions to the buffer, problems as small as $\frac{1}{8}^{th}$ the original size will incur a time penalty of at most a factor of 2. For most values of k, the penalty will be even less as illustated in Fig. 4.1 below, which is a typical profile of the performance degradation factor versus problem size for the case of the buffer split into four partitions.

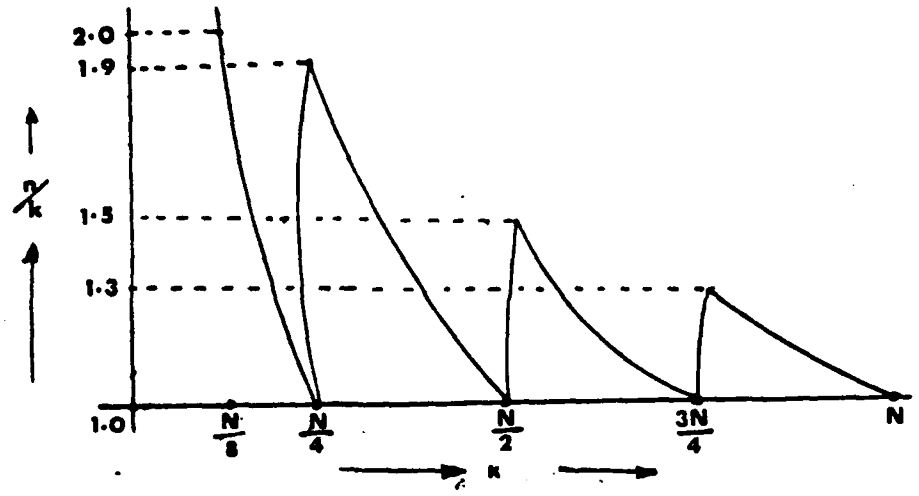

Figure 4·1

An ideal solution to small problem sizes is to design an algorithm on an array where the storage in any cell is independent of the problem size. Recently, we have been able to do this for matrix multiplication [13]. We are currently investigating algorithms for both these problems that can run on a linear array where the storage in any cell can be made independent of the problem size.

References

[1] A.V. Aho, J. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, (1974).

[2] R. Aubusson, and I. Catt, "Wafer-Scale Integration - A Fault Tolerant Procedure," *IEEE Journal of Solid-State Circuits*, SC-13 (3), (June, 1973), pp. 339-344.

[3] K.E. Batcher, "Design of a Massively Parallel Processor," *IEEE-TC*, Vol. C-9, No. 9, (September, 1980), pp. 836-840.

[4] K.Q. Brown, "Dynamic Programming in Computer Science," CMU Tech. Report (February, 1979).

[5] A.L. Fisher, and H.T. Kung, "Synchronizing Large VLSI Processor Arrays," *Proceedings of the Tenth Annual IEEE/ACM Symposium on Computer Architecture*, (June, 1983), pp. 54-58.

[6] L.J. Guibas, H.T. Kung, and C.D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," *Proceedings of the Conference on Very Large Scale Integration: Architecture, Design, Fabrication*, California Institute of Technology, (January, 1979), pp. 509-525.

[7] K.S. Hedlund, and L. Snyder, "Wafer Scale Integration of Configurable, Highly Parallel (CHiP) Processors (Extended Abstract)," *Proceedings of the 1982 International Conference on Parallel Processing*, (August, 1982), pp. 262-264.

[8] D.E. Knuth, *The Art of Computer Programming*, Vol. 3, Sorting and Searching, Addison-Wesley (1973).

[9] H.T. Kung, "Why Systolic Architectures," *IEEE Computer* 15(1), (January, 1980), pp. 37-46.

[10] F.T. Leighton, and C.E. Leiserson, "Wafer-Scale Integration of Systolic Arrays," *Proceedings of the Twenty-third Symposium on Foundations of Computer Science*, (November, 1982), pp. 297-311.

[11] K.N. Levitt, and W.H. Kautz, "Cellular Arrays for the Solution of Graph Problems," *CACM*, Vol. 15, No. 9, (1972), pp. 789-801.

[12] V.Y. Pan, "An Introduction to the Trilinear Technique of Aggregating, Uniting and Cancelling and Applications of the Technique for Constructing Fast Algorithms for Matrix Operations," *Proceedings of the Nineteenth Annual Symposium on Foundations of Computer Science*, (November, 1978).

[13] I.V. Ramakrishnan, and P.J. Varman, "Modular Matrix Multiplication on a Linear Array," Tech. Report - CS-TR-1340, Department of Computer Science, University of Maryland at College Park, (November, 1983).

[14] L. Snyder, "Introduction to the Configurable, Highly Parallel Computer," *Computer*, Vol. 15. No. 1, (January, 1982), pp. 47-56.

[15] S.J. Stolfo, and D.E. Shaw, "DADO: A Tree-Structured Machine Architecture for Production Systems," *Proceedings of AAAI*, (1982).

[16] J.D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, (1983).

[17] F.L. Van Scoy, "The Parallel Recognition of Classes of Graphs," *IEEE-TC*, Vol. C-29, No. 7, (July, 1980), pp. 563-570.

[18] P.J. Varman, and D.S. Fussell, "Design of Robust Systolic Algorithms," *Proceedings of the 1983 International Conference on Parallel Processing*, (August, 1983), pp. 458-460.

[19] S.W. Warren Jr., "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations," *CACM*, Vol. 18, No. 4, (April, 1975), pp. 218-220.

[20] S. Warshall, "A Theorem on Boolean Matrices," *JACM*, Vol. 9, No. 1, (January, 1972), pp. 11-12.

Appendix

We now provide proofs of correctness for the dynamic programming and transitive closure algorithms. We will also show that the area required by the array for these two algorithms is optimal. In the proofs to follow, in any reference to a control token we will assume that its control bit is set.

A. Proof of the Dynamic Programming Algorithm

We first establish that the algorithm described in Section 2 correctly computes $c(1,n+1)$. Let $\gamma = j-i$ and $\delta = n-i$, $1 \leq i < j \leq n+1$. In the following Lemma we establish the time at which $c(i,j)$ is transferred onto H_f and V_f.

Lemma A.1: $c(i,j)$ is transferred onto H_f and V_f in cell γ at time $2[\delta n+1+2(\gamma-1)]$.

Proof: $c(i,j)$ will be transferred onto H_f and V_f only if there is a control token present on H_c at cell γ's input at time $2[\delta n+1+2(\gamma-1)]$. This means that this control token must have been inserted at H_c of cell 1 at time $2[\delta n+1+2(\gamma-1)]-4(\gamma-1)$ (the elements in H_c encounter a delay of 4 cycle/cell). By step (2a) of the algorithm, a control token is inserted into the array on H_c at time $2[kn+1]$, $\forall k \geq 0$. $\square$

Lemma A.2: (1) $c(i,j)$ travels on H_f for 2γ cycles and is then transferred onto H_s in cell 2γ, and (2) $c(i,j)$ travels on V_f for $2(n+1)\gamma$ cycles and is then transferred onto V_s in cell 2γ.

Proof: We will prove (1) as the proof for (2) can be established along similar lines. By Lemma A.1, $c(i,j)$ is transferred onto H_f at time $t_1 = 2[\delta n+1+2(\gamma-1)]$. In 2γ additional cycles it will reach cell 2γ (delay on H_f is 2 cycles/cell).

In order for c(i,j) to be transferred onto H_s at cell 2γ, it must meet a control token on V_c at the input of 2γ at time $t_1+2\gamma$. This means that this control token must have been inserted into the array at time $t_3=t_1+2\gamma-(2\gamma-1)[2(n+1)+1]$ where $2(n+1)+1$ is the delay/cell encountered by control tokens on V_c. Substituting $\delta=n-i$ and $\gamma=j-i$, t_3 reduces to $2n[n-2(j-i)+1-i]+1$. Now $2\gamma\leq n$ as there are only n cells. So $2(j-i)\leq n$. Also $i\leq n$ and hence $[n-2(j-i)+1-i]\geq -n$. From step (2b) of the algorithm, a control token is inserted into the array on V_c at time $2kn+1$, $\forall k\geq -n$. $\square$

We are now ready to establish our main result about the correctness of computing c(i,j).

Theorem A.1: $c(i,j)=w_{ij}+\min\limits_{i<k<j}\{c(i,k)+c(k,j)\}$ when it is transferred onto H_f and V_f.

Proof: We prove this by *induction on* $\gamma=j-i$.

Basis. $\gamma=1$. The correct value of c(i,j) when $\gamma=1$ is its initial value w(i,j) which is stored in location δ of cell 1. At time $2\delta n+2$, address δ and a control token are inserted on the address belt and H_c respectively. So w(i,j) gets transferred onto H_f and V_f.

Inductive Step. We have to show that the Theorem holds $\forall i'$ and $\forall j'$ such that $j'-i'=\gamma+1$. Let $i'=i+\alpha-1$ and $j'=\alpha+j$. We will then have to show that $c(i+\alpha-1, \alpha+j)=\min\limits_{i+\alpha-1<k<\alpha+j}\{c(i+\alpha-1,k)+c(k,\alpha+j)\}$. To show this we must show the following.

1. $c(i+\alpha-1,k)$ and $c(k,\alpha+j)$ meet at cell $\gamma+1$ before $c(i+\alpha-1, \alpha+j)$ is transferred, and
2. when they meet, the address on the address belt at the input of $\gamma+1$ is $n-i-\alpha+1$.

By the inductive hypothesis and Lemma A.1, $c(i+\alpha-1,k)$ is correctly computed when it is transferred onto H_f and V_f at cell $k-i-\alpha+1$ at time $t_1=2[(n-i-\alpha+1)n+1+2(k-i-\alpha)]$. It then travels on H_f for an additional $2(k-i-\alpha+1)$ cycles. Subsequently, it travels on H_s till it reaches cell $\gamma+1$. Let t_2 denote the time taken to reach cell $\gamma+1$ after transfer. Now $t_2=\{2(k-i-\alpha+1)\}+[4(\gamma+1-2k+2i+2\alpha-2)]$. The expression within { } is the time it travels on H_f and that within [ ] is the time it travels on H_s. t_2 can be simplified to $2[i+2j+3\alpha-3k-1]$. So,

$$t_1 + t_2=2[(n-i-\alpha+1)n+1+2(k-i-\alpha)-3k+3\alpha+i+2j-1]$$
$$=2[(n-i-\alpha+1)n-k+\alpha-i+2j] \ldots (*)$$

By the inductive hypothesis again, $c(k,\alpha+j)$ is correctly computed in cell $\alpha+j-k$ when it is transferred onto H_f and V_f at time $t_3=2[(n-k)n+1+2(\alpha+j-k-1)]$. It then travels on V_f till it reaches cell $\gamma+1$. Let t_4 denote this travel time. So $t_4=2(n+1)(\gamma+1-\alpha-j+k)$ (recall that delay/cell on V_f is $2(n+1)$ clock cycles). Now $t_3 + t_4$ can be simplified to $2[(n-i-\alpha+1)n-k+\alpha-i+2j]$ which is the same as (*).

We will next show that $(*)\leq$ time at which $c(i+\alpha-1, \alpha+j)$ is transferred onto H_f and V_f. By Lemma A.1, this time is $2[(n-i-\alpha+1)n+1+2(j-i)]$. We then have to show that $2[(n-i-\alpha+1)n+1+2(j-i)]\geq 2[(n-i-\alpha+1)n-k+\alpha-i+2j]$ which reduces to showing that $-k+\alpha-i+2j\leq 1+2(j-i)$ and this is true as $i+\alpha-1<k<\alpha+j$.

In the proof we had assumed that $c(i+\alpha-1,k)$ travels on H_f and H_s whereas $c(k,\alpha+j)$ travels on V_f alone. We can also show in the symmetric case where $c(i+\alpha-1,k)$ travels on H_f and $c(k, \alpha+j)$ travels on V_f followed by V_s that they still meet at cell $\gamma+1$.

Lastly, we must show that the address on the address input at cell $\gamma+1$ is $n-i-\alpha+1$. This address must have been inserted on the address belt A_d of cell 1 at time $(*)-2(j-i)$ which can be simplified to $2\{(n-i-\alpha+1)n+1+[\alpha+j-1-k]\}$. From step (2c) of the algorithm, this address is $n-i-\alpha+1$ if $0\leq[\alpha+j-1-k]\leq n$. Now $1<k< \alpha+j \leq n+1$ and so $0<\alpha+j-k$

and hence $0 \leq \alpha+j-k-1$. Also $\alpha+j \leq n+1$ and hence $-k-1+\alpha+j \leq n$ as $k>0$ $\square$

B. Proof of the Transitive Closure Algorithm

We establish that after three passes the c(i,j)'s contain the transitive closure A^*. Our proof is along similar lines to the proof in [16] for the mesh-array algorithm in [6].

Recall that a k-path from vertex i to vertex j denotes a path from i to j that goes through no vertex numbered higher than k except the endpoints. Consequently, i and/or j may exceed k.

In the proofs that follow, the expression within { } will denote the time at which the elements are inserted in the array and that within [ ] will denote the time it takes to reach a cell after insertion.

Lemma B.1: In any pass a_{ik} and a'_{kj} meet at cell $i+j-1$.

Proof: a_{ik} reaches cell $i+j-1$ at time $t_1=\{t_s^p+n(n-1)+(i-1)n+k-1\}+[i+j-2]$. Similarly a'_{kj} reaches cell $i+j-1$ at time $t_2=\{t_s^p+(n-j)n+(k-1)\}+[(i+j-2)(n+1)]$. Now a'_{kj} travels at a delay of $(n+1)$/cell on V_f and hence $(i+j-2)$ is multiplied by a factor $(n+1)$ in t_2. The expression in t_2 can be simplified to $t_s^p+n(n-1)+(i-1)n+(k-1)+(i+j-2)$ which is the same as t_1. $\square$

Lemma B.2: In any pass, (1) a_{ij} and a'_{jj} meet at cell $i+j-1$, and a'_{ij} and a_{ii} also meet at cell $i+j-1$.

Proof: We will prove (1) and the proof for (2) is similar. Now a_{ij} arrives at cell $i+j-1$ at time $t_1=\{t_s^p+n(n-1)+(i-1)n+j-1\ \}+[i+j-2]$ and a'_{jj} arrives there at time $t_2=\{t_s^p+(n-j)n+(j-1)\}+[(i+j-2)(n+1)]$ which can be simplified and shown to be the same as t_1.

$\square$

Corollary B.1: a_{ij} and a_{ij}' are updated to the value of c(i,j) when they pass cell i+j-1.

Proof: a_{ij} and address i are inserted at the same time in the array. They both travel at the same speed and hence reach cell i+j-1 at the same time. A control token is inserted on V_c along with a_{jj}. They both travel at the same speed and hence when a_{ij} meets a_{jj}' it gets updated. A similar argument will prove that a_{ij} is also updated. $\square$

Lemma B.3: In any pass, (1) a_{ij} reaches cell i+k-1 at time $t_s^p + n(n-1)+(i-1)n + (j-1)+(i+k-2)$, and (2) a_{ij}' reaches cell k+j-1 at time $t_s^p + n(n-1)+(k-1)n+(j-1)+(i+k-2)$.

Proof: Immediate from steps (2a) and (2b) of the algorithm. $\square$

Lemma B.4: Suppose there is a min(i-j)-path from i to j, that is, a path that goes through no vertex as high as its end points. Then on pass 1 of the algorithm, a_{ij}, a_{ij}' and c(i,j) are all set to 1 at or before a_{ij} and a_{ij}' reach cell i+j-1.

Proof: We prove this by induction on the length of the shortest path from i to j. For the basis, paths of length 0 or 1, the Lemma holds as a_{ij} and a_{ij}' are 1 initially and c(i,j) is assigned 1 when either a_{ij} or a_{jj}', whichever reaches cell i+j-1 earlier.

For the induction, suppose there is a min(i,j)-path of length two or more from i to j. Then there exists some other vertex l on the path. Let l be the highest numbered vertex on this path. Now $l<i$ and $l<j$ because the path is a min(i,j)-path. Since l exceeds any other vertex on this path, there is a min(i,l)-path from i to l and a min(l,j)-path from l to j, and both of these paths are shorter than the path from i to j.

By the inductive hypothesis a_{il} and a_{lj}' are set to 1 at or before a_{il} reaches cell i+l-1 and a_{lj}' reaches cell l+j-1 respectively. Let t_1 and t_2 be the times when a_{il} and a_{lj} reach cell i+l-1 and l+j-1 respectively. From Lemma B.3, $t_1 = t_s^1 + n(n-1)+(i-1)n+(l-$

1)+(i+l-2), and $t_2=t_s^1+n(n-1)+(l-1)n+(l-1)+(l+l-2)$.

Let t_3 be the time at which they meet in cell i+j-1. Now $t_3=\{t_s^1+n(n-1)+(i-1)n+l-1\}+[i+j-2]$. As $i>l$ and $j>l$, $t_3>t_1$ and $t_3>t_2$. Recall that address i is inserted into the array along with a_{il} (step 2(e) of the algorithm). Consequently c_{ij} is assigned 1.

Let t_4 be the minimum of the time taken by a_{ij} and a_{ij}' to reach cell i+j-1 and so $t_4=t_s^p+n(n-1)+(i-1)n+\min(i,j)-1+(i+j-2)$. $i>l$ and $j>l$ and so $t_4>t_3$. Hence a_{ij} and a_{ij}' are assigned 1 when they reach cell i+j-1. $\square$

Lemma B.5: After pass 2 of the algorithm,

a. If there is a j-path from i to j, then c(i,j) and a_{ij} are set to 1 by time $t_s^p+n(n-1)+(i-1)n+(j-1)+(i+j-2)$.

b. If there is an i-path from i to j, then c(i,j) and a_{ij}' are set to 1 by time $t_s^p+n(n-1)+(i-1)n+(i-1)+(i+j-2)$.

c. If there is a max(i,j)-path from i to j then c(i,j) is set to 1 at some time.

Proof: We prove this by induction on the path length. If the length is 1 then $a_{ij}(a_{ij}')$ must be 1 if there is a j-path (i-path) from i to j. Hence c(i,j) will be assigned 1 when a_{ij} or a_{ij}' reaches cell i+j-1.

For the induction, suppose there is a j-path of length at least two from i to j. Let l be the highest numbered vertex on the path. Then $l<j$ and there is a shorter l-path from i to l. By the inductive hypothesis, a_{il} is set to 1 by time $t_1=t_s^2+n(n-1)+(i-1)n+(l-1)+(i+l-2)$. Since l is chosen to be the highest numbered vertex on the j-path, there is a min(l,j)-path from l to j. By Lemma B.4, a_{lj}' is already 1 by end of pass 1. Thus at time $t_2=t_s^2+n(n-1)+(i-1)n+(l-1)+(i+j-2)$ which is later than t_1, a_{il} and a_{lj}' meet at cell i+j-1 at which time c(i,j) is set to 1. It can be easily verified that a_{ij} and a_{ij}' arrive

at cell i+j-1 later than t_2. Hence they too are assigned 1.

We have proved (a). A similar argument will establish (b) and these two together imply (c). $\square$

We are now ready to establish our main result.

Theorem B.1: After the third pass, c(i,j) is set to 1 if there is any path from i to j.

Proof: By Lemma B.4, if there is a max(i,j)-path from i to j then a_{ij} is already 1 after pass 2. Otherwise, the highest numbered vertex l on some path from i to j is larger than either i or j. This means that there is an l-path from i to l and an l-path from l to j. The l-paths from i to l and l to j are a max(i,l)-path and max(l,j)-path respectively by the maximality of l. By Lemma B.5, a_{il} and a'_{lj} are set to 1 by end of pass 2. They meet again in cell i+j-1 in pass 3 at which time c(i,j) is assigned 1. $\square$

C. Area-Optimality of the Dynamic Programming Algorithm

The recurrence used to compute the dynamic programming problem (see Section 2) can be rewritten as:

$c_{ij}^{(0)} = w_{ij}, \ 1 \leq i < j \leq n+1$

$c_{ij}^{(\text{final})} = c_{ij}^{(o)} + \min_{i<k<j} \{ c_{ik}^{(\text{final})} + c_{kj}^{(\text{final})} \}$

We will establish that the area required by the linear array to compute the recurrence is assymptotically optimal. We establish this result under the following assumptions.

1. Any special purpose machine (a *chip* in VLSI) that computes the value of $c_{ij}^{(\text{final})}$ must compute $c_{ik}^{(\text{final})}$, $c_{kj}^{(\text{final})}$, $(\forall\ i<k<j)$.

2. The comparison and addition operation requires non-zero time.

3. The only input/output done by the machine is to read w_{ij} and output $c_{ij}^{(final)}$ (that is, we do not allow partially updated c_{ij} to leave the machine and re-enter at a later time).

Definition C.1: c_{ij} is said to be assigned a value when either

a. w_{ij} enters the machine or

b. $c_{ik}^{(final)} + c_{kj}^{(final)}$ has been computed for some k.

Under these assumptions we will establish that $\Omega(n^2)$ is a lower bound on the storage required by formulating the evaluation of the recurrence used to compute the dynamic programming problem as a game played with colored tokens on a graph G constructed as follows.

Let $G=(V,E)$ where $V=\{V_{i,j} \mid 1 \leq i < j \leq n\}$, and $E=\{ (V_{i,j}, V_{i+1,j}) \mid 1 \leq i < j-1 < n\} \cup \{(V_{i,j}, V_{i,j+1}) \mid 1 \leq i < j < n\}$

Fig. C.1 below illustrates the graph for n=16.

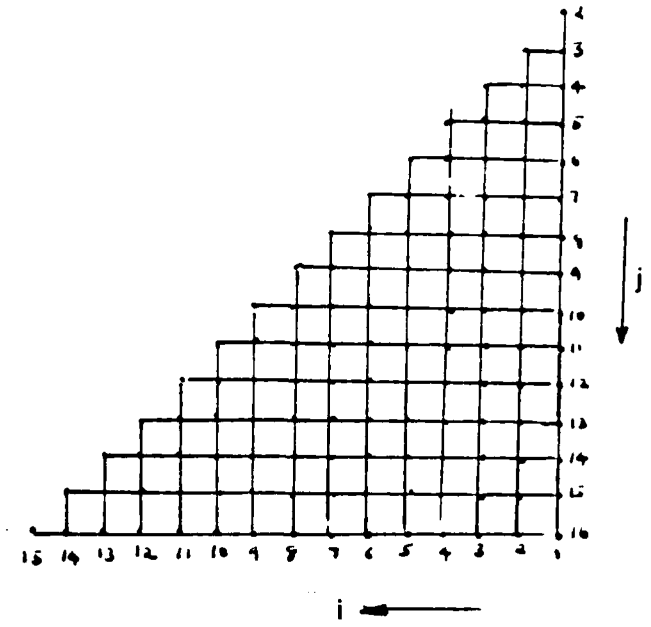

Figure C-1

The rules of the game are as follows.

1. Initially a *white* token is present on every vertex in V.

2. When c_{ij} is first assigned a value in the machine, the token on $V_{i,j}$ becomes *grey* in color.

3. When $c_{ij}^{(final)}$ leaves the machine, the token on $V_{i,j}$ becomes *black* in color.

4. Once a token changes color it cannot r turn to the color it had earlier.

5. All tokens change color from *white* to *grey* and finally to *black*. The computation is over when the token on $V_{i,n}$ becomes *black*.

6. Each token spends a non-zero amount of time when it is *grey*.

We introduce the following notations which will be used in the proofs.

Let $X \subseteq V$. A *column* of X is a subset of the vertices of X with the same first index. Similarly, a *row* of X is a subset of vertices of X with the same second index.

Let $X_w = \{ V_{i,j} \epsilon X \mid$ the token on $V_{i,j}$ is *white* $\}$, $X_g = \{ V_{i,j} \epsilon X \mid$ the token on $V_{i,j}$ is *grey* $\}$, and $X_b = \{ V_{i,j} \epsilon X \mid$ the token on $V_{i,j}$ is *black* $\}$.

Let A, B and C be three subsets of V defined as follows. $A = \left\{ V_{i,j} \epsilon V \mid \frac{n}{4} < i < j, \frac{n}{4} + 1 < j \leq \frac{3n}{4} \right\}$; $B = \{ V_{i,j} \epsilon V \mid 1 \leq i \leq \frac{n}{4}, \frac{n}{4} < j \leq \frac{3n}{4} \}$, and $C = \left\{ V_{i,j} \epsilon V \mid 1 \leq i \leq \frac{3n}{4}, \frac{3n}{4} < j \leq n \right\}$.

Fig C.2 illustrates the three subsets when n=16.

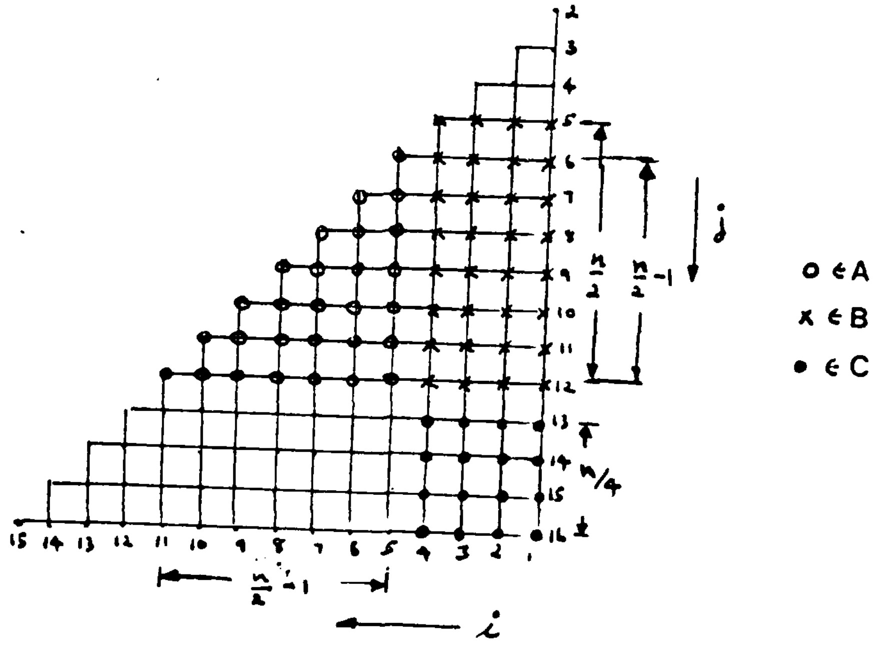

Figure C.2

For convenience, we will assume that n is a multiple of four. Let t' denote the time at which $c_{\frac{n}{4}+1,\frac{3n}{4}}$ obtains its final value $c^{(final)}_{\frac{n}{4}+1,\frac{3n}{4}}$. Now choose $t<t'$ to be the time at which $c^{(final)}_{\frac{n}{4}+1,k}$ and $c^{(final)}_{k,\frac{3n}{4}}$ $(\forall\ \frac{n}{4}+1<k<\frac{3n}{4})$ have been computed but $c^{(final)}_{\frac{n}{4}+1,\frac{3n}{4}}$ has been partially computed (that is, final assignment has not yet been made). From the recurrence relation it is seen that such a time instant must occur by assumption 2.

We will obtain a lower bound on the number of *grey* tokens on vertices in V at time t. Since a grey token corresponds to a c_{ij} value that is in the machine, by assuming that each such value requires unit storage, we will obtain the desired lower bound on the storage.

Lemma C.1: If the token on any $V_{i,j}\epsilon V$ is *white* then the token on any $V_{i,t}$ $(t<j)$ and any $V_{s,j}$ $(s>i)$ must be either *white* or *grey*.

Proof: If the token on $V_{i,j}$ is white then c_{ij} has not been assigned a value. Computing $c_{ij}^{(final)}$ requires the values of $c_{i,t}^{(final)}$ $(\forall\ j>t)$ and $c_{s,j}^{(final)}$ $(\forall\ s>i)$. Hence, none of these could have left the machine. Therefore, the tokens on $V_{i,t}$ $(t<j)$ and $V_{s,j}$ $(s>i)$ cannot be *black*. $\square$

Lemma C.2: At time t, (a) $C_b=\Phi$ and (b) $A_w=\Phi$.

Proof: (a) At time t, $c^{(final)}_{\frac{n}{4}+1,\frac{3n}{4}}$ has been partially computed. Suppose $V_{x,y}\epsilon C$ has a *black* token on it at time t. Since $c_{x,y}^{(final)}$ requires $c^{(final)}_{\frac{n}{4}+1,y}$ for computation and $c^{(final)}_{\frac{n}{4}+1,y}$ requires $c^{(final)}_{\frac{n}{4}+1,\frac{3n}{4}}$ for computation, this implies that $c^{(final)}_{\frac{n}{4}+1,\frac{3n}{4}}$ has already been computed -- a contradiction.

(b) At time t, $c^{(final)}_{s,\frac{3n}{4}}$ $(s>\frac{n}{4}+1)$ has been computed. Suppose $V_{x,y}\epsilon A$ $(x>\frac{n}{4}+1)$ had a *white* token on it. Then $c^{(final)}_{x,\frac{3n}{4}}$ could not have been computed -- a contradiction. Since $c_{\frac{n}{4}+1,\frac{3n}{4}}$ has been assigned a value, the token on $V_{\frac{n}{4}+1,\frac{3n}{4}}$ must be *grey*. Finally, since $c^{(final)}_{\frac{n}{4}+1,t}$ $(t<\frac{3n}{4})$ have been computed, all $V_{\frac{n}{4}+1,t}\epsilon A(t<\frac{3n}{4})$ must have a *grey* or *black* token. Hence $A_w=\Phi$. $\square$

Lemma C.3: If $|C_w|\geq\frac{n^2}{32}$ then $|B_g|+|B_w|\geq\frac{n^2}{16}$

Proof: Since $|C_w|\geq\frac{n^2}{32}$ at least $\frac{n}{8}$ columns of C have a *white* token. (A column is said to have a *white* token if the token on at least one vertex in the column is *white*). Then by Lemma C.1, at least $\frac{n}{8}$ columns of B must not have a *black* tokens. Thus at least $\frac{n}{8}\times\frac{n}{2}$ of the vertices in B have either *grey* or *white* tokens on them and hence, $|B_g|+|B_w|\geq\frac{n^2}{16}$. $\square$

Lemma C.4: If $|B_w|\geq\frac{n^2}{32}$ then $|A_w|+|A_g|\geq\frac{n^2}{512}$.

Proof: Since $|B_w|\geq\frac{n^2}{32}$, at least $\frac{n}{16}$ rows of B must have a *white* token. By Lemma C.1, at least $\frac{n}{16}$ rows of A must not have a *black* token. Thus, at least $\frac{1}{2}\times\frac{n}{16}\times\frac{n}{16}=\frac{n^2}{512}$ of the vertices in A must have *grey* or *white* tokens, that is $|A_w|+|A_g|\geq\frac{n^2}{512}$. $\square$

Theorem C.1: At time t, the number N of *grey* tokens on vertices in V is $\Omega(n^2)$.

Proof: Since $|C| = \frac{n^2}{16}$, by Lemma C.2, it follows that at time t, $|C_g| + |C_w| = \frac{n^2}{16}$. Thus, at least one of the following must hold: $|C_g| \geq \frac{n^2}{32}$ or $|C_w| \geq \frac{n^2}{32}$. If $|C_g| \geq \frac{n^2}{32}$ then $N=\Omega(n^2)$. If $|C_w| \geq \frac{n^2}{32}$ then by Lemma C.3, $|B_g| + |B_w| \geq \frac{n^2}{16}$. Again, at least one of the following two conditions must hold: $|B_g| \geq \frac{n^2}{32}$ or $|B_w| \geq \frac{n^2}{32}$. If $|B_g| \geq \frac{n^2}{32}$ then $N=\Omega(n^2)$. If $|B_w| \geq \frac{n^2}{32}$, then by Lemma C.4, $|A_w| + |A_g| \geq \frac{n^2}{512}$. By Lemma C.2 however, at time t, $|A_w| = \Phi$ and thus, $|A_g| \geq \frac{n^2}{512}$. Hence, in all cases $N=\Omega(n^2)$. □

D. Area-Optimality of the Transitive Closure Algorithm

We will now establish that the area required by the transitive closure algorithm is optimal. We obtain a lower bound on the storage required to compute matrix multiplication. As matrix-multiplication and transitive closure are related [16] the lower bounds on the area are the same to within a constant factor.

Let a_{ij}, b_{ij}, and c_{ij} denote the (ij)th element in matrix A, matrix B and result matrix C respectively. We establish this result under the following assumptions:

1. Any special-purpose machine (like a linear array) that multiplies matrices A and B must compute $a_{ik}b_{kj}$ ($\forall i$, $\forall j$ and $\forall k$ $|1 \leq i,j,k \leq n$).

2. The special-purpose machine has a constant number of I/O ports.

3. The elements of the matrices A, B and C are inserted into the special-purpose machine <u>only once</u> through the input ports.

Under these assumptions we will establish that $\Omega(n^2)$ is a lower bound on the storage that is required by any special-purpose machine that multiplies two $n \times n$ matrices. We obtain this bound by formulating the computation of matrix multiplication as a game played with tokens on an undirected graph constructed as follows:

Let $G_k=(V_k, E_k)$, $k=1,..,n$ where

$V_k=\{f_{ik}, h_{kj} \mid i=1,..n \text{ and } j=1,..,n\}$ and

$E_k=\{<f_{ik}, h_{kj}> \mid i=1,..,n \text{ and } j=1,..,n\}$

The rules of the game are as follows:

1. A token is placed on f_{ik} (h_{kj}) when a_{ik} (b_{kj}) is inserted into the machine.
2. Updating c_{ij} (by adding $a_{ik}b_{kj}$ to c_{ij} for some k) results in removing the edge $<f_{ik}, h_{kj}>$ from G_k.
3. An edge is removable only if there are tokens at both end vertices.
4. A token from a vertex is removable only if all the edges incident on the vertex are removable. When a token from a vertex is removed then all the incident edges on the vertex are deleted. (The token will eventually leave the machine and will *never* reenter.)

We will assume that each token occupies unit storage ($O(1)$). We also assume that a partially updated c_{ij} also occupies unit storage. (At any instant of time c_{ij} is partially updated if there exists some k ($1 \leq k \leq n$) such that $a_{ik}b_{kj}$ either has not been computed and/or added to c_{ij} by that time instant .)

Let x_k be the earliest time at which the first token in G_k is removable and let y_k be the earliest time at which all the tokens in G_k are removable. Since only a constant number of tokens enter the machine at any time, by choosing n sufficiently large, we can ensure that $\forall k\ (1 \leq k \leq n)\ x_k < y_k$. $\forall k\ (1 \leq k \leq n)$, let $I_k = [x_k, y_k]$ denote the time interval between and including x_k and y_k.

Lemma D.1: At any time t such that $x_k \leq t < y_k$, there are at least n tokens in G_k.

Proof: Without any loss of generality, let the first (or one of the first if there are more than one) token(s) that can be removed from G_k be the one on vertex f_{mk}. At $t_1 = x_k$, then, there must be tokens on all h_{kj} $(1 \leq j \leq n)$. We claim that no token on any h_{kj} will be *removable* at any t $(x_k \leq t < y_k)$.

Assume this is not the case, and at $t < y_k$, let h_{kj} be the first vertex (or one of the first vertices) from which a token is removable. This implies that there must be tokens on *all* vertices f_{jk} that still have incident edges. This means that all the edges still remaining in G_k are removable, and consequently all the remaining tokens in G_k are removable at time t. But then $t = y_k$ -- a contradiction. Hence no token on any h_{kj} is removable at any time t $(x_k \leq t < y_k)$. Each h_{kj} has a token and hence the Lemma.

□

Lemma D.2: Let $m < n$. For any i, if $t \geq y_i$ and G_i has m tokens then at least $\frac{n^2}{2}$ edges must have been deleted from G_i.

Proof: There are m tokens in G_i. Since $t \geq y_i$, the absence of a token on a vertex means that all the n edges incident on the vertex have been deleted. (At $t = y_i$, all edges in G_i are removable). The number of absent tokens$=2n-m$ which is greater than n as $m < n$.

Now one edge is in common with at most two vertices. Thus the 2n-m absent tokens result in at least $\frac{n^2}{2}$ deleted edges. $\square$

Let us impose an ordering on the sets I_k such that $x_{i_1} \leq x_{i_2} \leq .. \leq x_{i_n}$ and let $\Gamma =$ $I_k \mid y_k \leq x_{i_n}$ and $\Lambda = \{I_k \mid y_k > x_{i_n}\}$.

Theorem D.1: Any matrix-multiplication machine requires $\Omega(n^2)$ storage.

Proof: Since $|\Gamma|+|\Lambda|=n$, either $|\Gamma| \geq \frac{n}{2}$ or $|\Lambda| \geq \frac{n}{2}$.

Case 1: $|\Lambda| \geq \frac{n}{2}$ (see Fig. D.1)

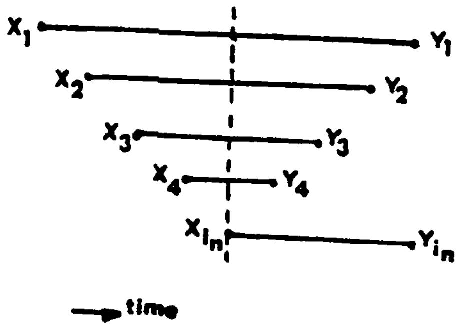

Figure D·1

At $t=x_{i_n}$ all the intervals in Λ satisfy Lemma D.1. Hence at $t=x_{i_n}$, there are at least $n(\frac{n}{2})$ tokens in the machine. So the storage required is $\Omega(n^2)$.

Case 2: $|\Gamma| \geq \frac{n}{2}$ (see Fig. D.2)

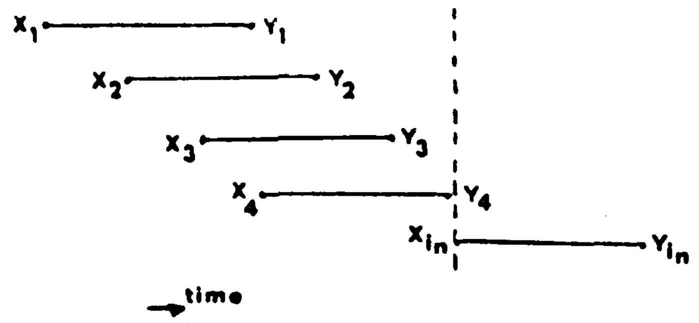

Figure D.2

At $t = x_{i_n}$, either all G_k, such that $I_k \in A$, have n tokens on them, or at least one of them has less than n tokens. If every G_k has n tokens then the storage required is again $\Omega(n^2)$. If any one, say G_r, has less then n tokens then by Lemma D.2 G_r must have released at least $\frac{n^2}{2}$ edges. Now each released edge corresponds to a partially updated c_{ij}. None of the c_{ij}'s could have left the machine as all of them are finally updated only at $t \geq x_{i_n}$. Thus at any time t $(y_k \leq t \leq x_{i_n})$ there are at least $\frac{n^2}{2}$ partially updated c_{ij}'s in the machine. The case $y_k = x_{i_n}$ is covered by assumption 2 which precludes the possibility of all these c_{ij}'s being instantaneously updated and leaving the machine. So the storage required for the partially updated c_{ij}'s must be $\Omega(n^2)$. $\square$